MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 816 December, 1984

PP

A Lisp Pretty Printing System

by

Richard C. Waters

ABSTRACT

The PP system provides an efficient implementation of the Common Lisp pretty printing function PPRINT. In addition, PP goes beyond ordinary pretty printers by providing mechanisms which allow the user to control the exact form of pretty printed output. This is done by extending Lisp in two ways. First, several new FORMAT directives are provided which support dynamic decisions about the placement of newlines based on the line width available for output. Second, the concept of print-self methods is extended so that it can be applied to lists as well as to objects which can receive messages. Together, these extensions support pretty printing of both programs and data structures.

The PP system also modifies the way that the Lisp printer handles the abbreviation of output. The traditional mechanisms for abbreviating lists based on nesting depth and length are extended so that they automatically apply to every kind of structure without the user having to take any explicit action when writing print-self methods. A new abbreviation mechanism is introduced which can be used to limit the total number of lines printed.

Keywords: Pretty Printing, Formatted Output, Abbreviated Output, Lisp

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505, and in part by National Science Foundation grants MCS-7912179 and MCS-8117633.

The views and conclusions contained in this document are those of the author, and should not be interpreted as representing the policies, neither expressed nor implied, of the Department of Defense, nor of the National Science Foundation.

I - Introduction

The main purpose of this paper is to serve as a manual for the Lisp Machine implementation of the PP system (which can be used by loading the file "PP" from the LMLIB directory). PP has been written to run under releases 4 and 5 of the Symbolics Lisp Machine system [4] and the documentation in this paper is directed toward these releases. PP will be converted to run in other releases of the Lisp Machine system as they appear. This conversion will lead to some minor changes (as indicated below) in the user interface due to the introduction of Common Lisp compatibility into Lisp Machine Lisp. In particular, several output control variables have different names in Common Lisp.

The relevance of the PP system is not restricted solely to Lisp Machine Lisp. The system has been designed to be upward compatible with Common Lisp and could be implemented with little modification in most Lisp dialects. Going beyond this, many of the ideas discussed in this paper have a much wider area of applicability. For example, the mechanisms for allowing the user to exercise control over the dynamic arrangement of output could be incorporated into almost any programming language (e.g., into the formatted output statements of Fortran or PLI).

The PP system provides three sets of features. First, as described in Section II, PP supports several of the Common Lisp output functions and output control variables. In particular, it provides an efficient implementation of the pretty printing function PPRINT. Second, PP goes beyond typical pretty printing systems by providing a flexible interface which gives the user detailed control over how program structures and data structures are to be pretty printed. This is done by extending the standard Lisp concepts of FORMAT control strings (see Section III), and print-self methods (see Section IV). Third, as described in Section V, the way the abbreviation of output is handled by Lisp is modified and extended.

The PP system is based on the earlier GPRINT system [5,6]. The main difference between the two systems is that PP has a greatly improved user interface. People who have used GPRINT will recognize that most of the features of PP descend from features of GPRINT, however this paper assumes no knowledge of GPRINT.

The pretty printing algorithm used by PP is essentially the same as the one used by GPRINT, though it has been streamlined and made considerably more efficient. This paper does not go into a detailed discussion of the algorithm since it is fully discussed elsewhere (see [2,5]). However, it should be noted that a key aspect of the algorithm is that it is an inherently fast linear algorithm which uses very little storage. As a result, pretty printing is not significantly slower than ordinary printing.

II - Basic Output Functions and Control Variables

The PP system supports several of the Common Lisp output functions and control variables which are not supported by release 5 of Lisp Machine Lisp. These functions and variables are discussed in detail in the Common Lisp documentation [3] and summarized below.

Pretty Printing

The variable *PRINT-PRETTY* (which has a default value of NIL) controls pretty printing. When this variable is non-NIL it causes all of the standard output functions (PRIN1, PRINC, etc.) to perform pretty printing. The function PPRINT binds *PRINT-PRETTY* to T while printing its argument and therefore forces pretty printing.

PPRINT *object* &optional *stream*

This is the same as the function PRINT except that it binds *PRINT-PRETTY* to T, does not print a trailing space, and returns no values.

Abbreviation

The PP system supports length and depth abbreviation under control of the variables PRINLENGTH and PRINLEVEL. (When Common Lisp compatibility is introduced these variables will be called *PRINT-LENGTH* and *PRINT-LEVEL*). This abbreviation is supported both when pretty printing and when not pretty printing. As discussed in Section V, the way in which this abbreviation is handled is modified in order to make it more convenient to use. In addition, a new kind of abbreviation is supported which can be used to limit the total number of lines printed. As discussed in Section V, this abbreviation is controlled by the variable PP:*PRINT-LINES*.

The Printing of Escape Characters

The variable *PRINT-ESCAPE* (which has a default value of T) controls the printing of escape characters. The function PRINC binds *PRINT-ESCAPE* to NIL. Other output functions (e.g., PRINT and PPRINT) bind *PRINT-ESCAPE* to T.

A Neutral Output Function

The most basic Common Lisp output function is the function WRITE. Unlike the other output functions, WRITE is neutral in that it does not force the specification of values for any of the output control variables. The function WRITE does, however take keyword arguments which, if supplied, specify values for output control variables. The implementation of WRITE provided by PP supports a new keyword :LINES which corresponds to the variable PP:*PRINT-LINES*.

WRITE *object* &key :stream :escape :base :pretty :level :length :lines :array
:radix :circle :case :gensym

This function is like PRIN1 except that it does not force *PRINT-ESCAPE* to be T. The :STREAM keyword argument (which defaults to STANDARD-OUTPUT) specifies the output stream. The other keyword arguments allow the specification of values for the control variables *PRINT-ESCAPE*, BASE, *PRINT-PRETTY*, PRINLEVEL, PRINLENGTH, PP:*PRINT-LINES*, SI:*PRINARRAY*, *PRINT-RADIX*, *PRINT-CIRCLE*, *PRINT-CASE*, and *PRINT-GENSYM* respectively. (The last four of these variables are not supported by release 5 of the Lisp Machine System.)

A Neutral Format Directive

PP provides a new FORMAT directive ~W which corresponds to calling the function WRITE without any keyword arguments -- i.e., in contrast to ~A and ~S it is neutral in that it obeys the value of *PRINT-ESCAPE* rather than specifying its value. The directive ~W takes the same parameters as ~A and ~S -- i.e., *~mincol, colinc, minpad, padchar*W. As with ~A and ~S the atsign flag causes spacing to be inserted on the left rather than on the right.

The colon flag has a different meaning from the one used for ~A and ~S. If the colon flag is used with ~W then several of the output control variables are given values which are useful when pretty printing is desired. Using the colon flag binds *PRINT-PRETTY* and *PRINT-ESCAPE* to T forcing pretty printing and the printing of escape characters and binds PRINLEVEL, PRINLENGTH, and PP:*PRINT-LINES* to NIL turning off abbreviation. The use of ~W with and without the colon flag is illustrated in the following example.

```
(LET ((*PRINT-PRETTY* NIL)
      (*PRINT-ESCAPE* NIL)
      (PRINLENGTH 2)
      (X '(QUOTE ("a" "b" "c"))))
  (FORMAT T "~W ~:W" X X))
```

The output produced by the example is shown below. Since *PRINT-PRETTY* and *PRINT-ESCAPE* are bound to NIL, and PRINLENGTH is bound to 2, the first use of ~W produces output which is not pretty printed and which is printed without escape characters, and which is abbreviated as shown. The use of ~:W forces pretty printing and the printing of escape characters, and suppresses the abbreviation.

```
(QUOTE (a b ...)) '("a" "b" "c")
```

If the LET bound *PRINT-PRETTY* and *PRINT-ESCAPE* to T, and PRINLENGTH to NIL, then both outputs would be the same.

A More Efficient Implementation of Format

As will become clear in the next section, the user interface to the pretty printing facilities provided by the PP system centers around the function FORMAT. This leads to a very convenient interface which builds on top of the standard FORMAT directives. Unfortunately, the function FORMAT is rather inefficient due to the fact that the control string is executed interpretively. In order to combat this problem, a macro is provided which creates efficient compiled code corresponding to a FORMAT control string.

PP:FAST-FORMAT *destination control-string* **&rest** *args*

This macro is the same as the function FORMAT except that *destination* is required to either be a literal NIL or a literal T or a stream -- it cannot be a string. PP:FAST-FORMAT is advantageous to use because, if the *control-string* is a sufficiently simple constant, the output will be performed significantly faster than it would be by FORMAT.

At the current time, PP:FAST-FORMAT is only capable of compiling relatively simple FORMAT control strings such as the ones used as examples in this paper. However, there is no reason why it could not be extended so that it could compile almost all possible FORMAT control strings.

In order to support rapid pretty printing of programs, PP internally uses the function PP:FAST-FORMAT instead of FORMAT wherever possible. This approximately triples the speed of the system.

Interactive Output Control Variables

In keeping with traditional Lisp practice, the style of output (e.g., whether escape characters should be printed and whether abbreviation should be used) is controlled by dynamic variables. One effect of this design decision is that if these variables are given particular top level values, these values become the default values for all output. Unfortunately, this is not always desirable. For example, one might well want to use abbreviation when printing the interactive output from the READ-EVAL-PRINT loop and yet not want to use abbreviation for lower level output inside programs (e.g., when writing to files, or strings). In this situation, one is forced to turn abbreviation on globally and then explicitly turn it off for each lower level call on an output function.

The PP system combats this problem by providing a mechanism for specifying a set of values for the output control variables which apply only to interactive output and not to other output. This mechanism is based on the special Lisp Machine variable PRIN1 (inherited from MacLisp) which specifies a printing function which the Lisp system will use for much of its interactive output to the user. For example, the Lisp READ-EVAL-PRINT loop uses the value of PRIN1 to display results to the user. (It should be noted that, as of release 5 of the Lisp Machine system, setting the value of the variable PRIN1 unfortunately does not succeed in controlling all of the interactive output to the user. For example, the debugger does not use the value of PRIN1 when displaying the contents of program variables in a stack frame.)

A function PP:INTERACTIVE-WRITE is provided which is intended to be used as the value of the variable PRIN1. A second function PP:SET-INTERACTIVE-CONTROL-VARIABLES is provided which controls the printing initiated by PP:INTERACTIVE-WRITE.

PP:INTERACTIVE-WRITE *object* &optional *stream*

This function is similar to the standard Common Lisp function WRITE except that it is sensitive to the control values specified by PP:SET-INTERACTIVE-CONTROL-VARIABLES rather than to the standard output control variables.

PP:SET-INTERACTIVE-CONTROL-VARIABLES &key :escape :base :pretty :level :length :lines
:array :radix :circle :case :gensym

This function specifies the output control variable values to be used by the function PP:INTERACTIVE-WRITE. The keyword arguments correspond to the control variables *PRINT-ESCAPE*, BASE, *PRINT-PRETTY*, PRINLEVEL, PRINLENGTH, SI:*PRINARRAY*, *PRINT-RADIX*, *PRINT-CIRCLE*, *PRINT-CASE*, and *PRINT-GENSYM* respectively. (The last four of these variables are not supported by release 5 of the Lisp Machine System.)

The recommended way to utilize this mechanism is to use the function PP:INTERACTIVE-WRITE as the value of the variable PRIN1 and set up interactive control variable values as shown below. Experience has shown the values illustrated (which are the default) to be particularly useful.

```
(SETQ PRIN1 #'PP:INTERACTIVE-WRITE)
(PP:SET-INTERACTIVE-CONTROL-VARIABLES ':ESCAPE T
                                      ':PRETTY T
                                      ':LEVEL NIL
                                      ':LENGTH NIL
                                      ':LINES 4)
```

III - Controlling the Arrangement of Output

By extending the concept of a FORMAT control string, the PP system makes it possible to give explicit directions for how something is to be pretty printed. The discussion below assumes that the reader has a basic understanding of the function FORMAT. The Lisp Machine documentation [4] describes FORMAT in detail.

Two basie concepts underlie the way PP supports the dynamic arrangement of output -- logical blocks and conditional newlines. The output as a whole is divided up hierarchically into logical blocks and sub-blocks. Each logical block is divided up into sections by positions where conditional newlines are specified. When printing the output, each logical block is printed on a single line if possible. If a logical block is too long to be printed on a single line and *PRINT-PRETTY* is T then newline characters are inserted at appropriate conditional newline positions in the block in order to print the block on several lines. The pretty printing algorithm uses internal buffering of the output so that it can determine which way to print a logical block before actually outputting any of the characters in the block.

As an example of how logical blocks and logical block sections can be used to control the arrangement of output, consider the following. Suppose that a user wants to print out the line of output shown below.

```
Roads ELM MAIN Towns BOSTON LOWELL
```

Suppose further that if there is not enough room to print the output on a single line then the user would like to have the output printed on two lines as follows.

```
Roads ELM MAIN
Towns BOSTON LOWELL
```

The user could achieve this goal by grouping the output as a whole into a logical block, and using a conditional newline to divide the block into two sections "Roads ELM MAIN ", and "Towns BOSTON LOWELL". The sections would be printed together on one line if there was room, and on separate lines if there was not. The way in which a user can specify logical blocks and conditional newlines is described in the next two subseetions.

Specifying Logical Blocks

The FORMAT directive ~I...~. is used to specify a logical block. In addition to specifying a logical block, the directive ~I...~. descends into the corresponding FORMAT argument (which must be a list) in the same way as the standard directive ~1{...~} (iterate [once] over list). If the corresponding format argument is not a list, then it is printed using ~W and the ~I...~. directive is skipped. The directive ~^ (terminate iteration) can be used to exit from a ~I...~. just as it can be used to exit from a ~1{...~}. The ~I...~. directives in the example below decompose the list argument and specify three logical blocks -- the output as a whole, the road names, and the town names.

```
(FORMAT T "~IRoads ~I~S ~S~. Towns ~I~S ~S~.~."
          '((ELM MAIN) (BOSTON LOWELL)))
```

Each logical block is printed in one of two ways: *single-line* mode or *multi-line* mode. The decision of which mode to use is made based on the value of *PRINT-PRETTY*, on how many characters are in the block, and on how much line width is available for printing the block.

The column position where a logical block must start (and therefore the line width available for printing it) is determined by the preceding output. If *PRINT-PRETTY* is T and the line width available is less than the size of the logical block then the block is printed in multi-line mode. Otherwise, single-line mode is used.

In single-line mode, all of the characters in a logical block are printed as a unit on the end of the current line. The call on FORMAT above will produce the output shown below when the logical blocks are printed in single-line mode.

```
Roads ELM MAIN Towns BOSTON LOWELL
```

If a logical block is printed in multi-line mode, then newlines are inserted in the block at positions specified by the user. (In the contrived example above, no such potential newline positions are specified. As a result, printing the blocks in multi-line mode would not cause any change in the output.)

The colon flag can be used with ~I...~. with the same meaning as with ~W -- i.e., *PRINT-PRETTY* and *PRINT-ESCAPE* are bound to T and PRINLEVEL, PRINLENGTH, and PP:*PRINT-LINES* are bound to NIL during the processing of the logical block.

If the atsign flag is used with ~I...~. then the directive operates on all of the rest of the arguments in much the same way as the standard directive ~1@{...~} (iterate [once] over arguments.) However, unlike ~1@{...~}, ~@I...~. always uses up all of the remaining arguments. The following example is identical to the one above except that it uses ~@I...~..

```
(FORMAT T "~@IRoads ~I~S ~S~. Towns ~I~S ~S~.~."
          '(ELM MAIN) '(BOSTON LOWELL))
```

Multi-Line Mode Conditional Newlines

The places in a logical block where newlines can be inserted when the block is printed in multi-line mode are called *conditional* newline positions. The most basic kind of conditional newline is called a *multi-line mode* conditional newline. When a logical block is printed in multi-line mode, a newline is inserted at every multi-line mode conditional newline which is immediately contained in the block. In order to specify a multi-line mode conditional newline, you use the FORMAT directive ~_ as shown in the example below. This example is the same as the one in the last subsection except that each of the three logical blocks is divided into two sections by a ~_ directive. If a ~_ directive is encountered outside of a logical block, it is ignored.

```
(FORMAT T "~:@IRoads ~I~S ~_~S~. ~_Towns ~I~S ~_~S~.~."
          '(ELM MAIN) '(BOSTON LOWELL))
```

If the line width available is at least 34, the top level logical block, and hence the whole output, will be printed in single-line mode as shown in the last subsection. If the line width available is only 30, the top level logical block will be printed in multi-line mode. As shown below, this causes the insertion of a newline at the multi-line mode conditional newline which is immediately contained in this block.

```
Roads ELM MAIN
Towns BOSTON LOWELL
```

If the line width available is only 15, the second logical sub-block will also be printed in multi-line mode as shown below. An important aspect of multi-line mode printing is that whenever a newline is inserted, the next line is indented so that it begins in the same column as the first character in the block.

```
Roads ELM MAIN
Towns BOSTON
      LOWELL
```

If the line width is less than 15, all three logical blocks will be printed in multi-line mode.

```
Roads ELM
      MAIN
Towns BOSTON
      LOWELL
```

Further reductions in the line width available would not alter the form of the output because the user has not specified any more places where newlines can be inserted.

Prefixes and Suffixes

The portion of a FORMAT control string enclosed in ~!...~. can be dividcd up into three segments by ~; as follows ~!*prefix*~;*body*~;*suffix*~.. If the enclosed portion is only dividcd into two segments then the suffix defaults to the null string. If thc portion consists of only a single segment (as in the examples above) thcn both the prefix and the suffix default to the null string. The prefix and suffix must both be constant strings i.e., they cannot contain FORMAT dircctives. The body can be any arbitrary FORMAT control string.

When a ~!*prefix*~;*body*~;*suffix*~. directive is processed, the prefix is printed out as thc initial characters in the logical block, and the suffix is printed out as the final characters in the logical block. This bchavior is the same as if the characters in thc prefix and suffix were simply uscd as the initial and final characters (respectively) of the body except for two things. First, whenever a ncwline is insertcd in a logical block, the ncxt line is indented so that it begins in the same column as the first character in the body of the block -- i.e., it is indcnted past the prcfix. Second, the suffix is always printcd out even if the processing of the body is cut short (e.g., by ~^ or by length abbreviation (see Section V)). The use of a prefix and suffix are illustrated in the example below.

```
(DEFUN PRINT-ADDRESS (ADDRESS)
  (FORMAT T "~:!#<~;Address - ~_~S~^ ~_~S~;>~." ADDRESS))
```

The function PRINT-ADDRESS prints a list representing a street address. If the expression (PRINT-ADDRESS '(256 MAPLE))) is evaluated with a linc length of 10 the following output is produced. Note that the house number and street name are indented so that thcy line up after the prefix "#<".

```
#<Address -
  256
  MAPLE>
```

If the cxpression (PRINT-ADDRESS '(MAPLE))) is evaluated with a line length of 10 the following output is produced. Note that, the suffix ">" is printed, even though the absence of a house number causes the ~^ dircctive to terminate processing of the body before the end of the body is reached.

```
#<Address -
  MAPLE>
```

Special Features of Prefixes and Suffixes

The ~!...~. directive can be given a parameter which controls the default values of the prefix and suffix. If the parameter is 0 (the default value) then the prefix and suffix both default to the null string as shown in the examples above. If the parameter is 1 then the prefix defaults to "(" and the suffix defaults to ")". The use of this parameter is illustrated in the example below.

```
(FORMAT T "~1:!~S ~_~S~." (ELM MAIN))
```

Assuming a line width of 8 the above call on FORMAT produces the following output.

```
(ELM
 MAIN)
```

If the atsign flag is used with the ~; directive that marks the end of the prefix in a ~!...~. directive then the prefix is printed before every line in the logical block, instead of just before the first line. This feature is illustrated in the example below.

```
(FERROR "~:@!;~@;Bound variable ~S is not a symbol in ~_~S.~."
        2 '(LET ((2 X-VALUE)) (+ X-VALUE X-VALUE)))
```

The call on FERROR produces the following output assuming a line length of 45.

```
>>Error: ;Bound variable 2 is not a symbol in
         ;(LET ((2 X-VALUE))
         ;  (+ X-VALUE X-VALUE)).
```

Whenever per-line prefixes are printed, the prefixes on the second and subsequent lines are indented so that all the prefixes line up. The call on FERROR also illustrates that the FORMAT directives provided by PP can be used in a FORMAT control string which is not directly used as an argument to the FORMAT function.

Nested logical blocks can each specify a per-line prefix as in the example below. All appropriate per-line prefixes are printed on each line at the appropriate indentation points.

```
(FORMAT T "~:@!;;; ~@;The error message will have the following form~_~
       >>Error: ~@!;~@;Bound variable ~S is not a symbol in ~_~S.~.~."
       2 '(LET ((2 X-VALUE)) (+ X-VALUE X-VALUE)))
```

The call on FORMAT produces the following output assuming a line length of 50.

```
;;; The error message will have the following form
;;; >>Error: ;Bound variable 2 is not a symbol in
;;;          ;(LET ((2 X-VALUE))
;;;          ;  (+ X-VALUE X-VALUE)).
```

If-Needed Conditional Newlines

PP supports a second kind of conditional newline (called an *if-needed* conditional newline) which allows more flexible control over where newlines are placed when a logical block is printed in multi-line mode. A newline is inserted at an if-needed conditional newline only if the *following section of output* is too long to fit on the end of the current line. Note that *the following section of output* is all of the output up to the next place immediately contained in the block where a newline can occur, or the end of the block, whichever comes first. The pretty printing algorithm uses internal buffering of the output in order to determine whether or not to insert a newline at an if-needed conditional newline position before actually outputting any of the characters in the following section of output. In order to specify an if-needed conditional newline, you use the FORMAT directive ~:_ as shown in the example below. If a ~:_ directive is encountered outside of a logical block, it is ignored.

```
(FORMAT T "Roads ~:!~@{~S~^ ~:_~}~." '(ELM MAIN MAPLE CENTER RIVER HIGH))
```

The FORMAT control string uses the standard directives ~@{...~} (iterate over arguments) and ~^ (terminate iteration) in order to decompose the list argument. Each element of the list is printed with ~S. A space and an if-needed conditional newline is placed after each element except the last.

If the line width available is at least 38, the call on FORMAT above will print on one line. If the line width is only 25, multi-line mode will be used as shown below. Only one newline is inserted because the if-needed conditional newlines in the block are processed one at a time and only when looking at the if-needed conditional newline after the third road name is it determined that the following section of the block (i.e., "CENTER ") cannot fit on the end of the line.

```
Roads ELM MAIN MAPLE
      CENTER RIVER HIGH
```

If the line width is only 20, newlines will be inserted at the if-needed conditional newlines after the second and fourth road names.

```
Roads ELM MAIN
      MAPLE CENTER
      RIVER HIGH
```

If the line width is less than 15, newlines will be inserted at all of the if-needed conditional newlines.

```
Roads ELM
      MAIN
      MAPLE
      CENTER
      RIVER
      HIGH
```

Filling

If-needed conditional newlines can be used when outputting text in order to make the text aesthetically fill the space available. For example, consider the call on FERROR used as an example above. The FORMAT control string specifies only one place where a newline can be inserted in the error message. As a result, a minimum line width of 45 is required to print it. The example below shows how the FORMAT control string can be modified so that the error message will print aesthetically on shorter line lengths.

```
(FERROR "~:@!;~@;Bound ~:_variable ~:_~S ~:_is ~:_not ~:_a ~:_~
                 symbol ~:_in ~:_~S.~."
        2 '(LET ((2 X-VALUE)) (+ X-VALUE X-VALUE)))
```

An if-needed conditional newline is placed between each pair of words in the error message. This causes the pretty printer to dynamically decide where to place newlines in order to cause the error message to fill the line width available. If the modified call on FERROR is evaluated with a line length of 30 then the following output is produced.

```
>>Error: ;Bound variable 2 is
         ;not a symbol in
         ;(LET ((2 X-VALUE))
         ;  (+ X-VALUE
         ;     X-VALUE)).
```

FORMAT control strings like the one above are very useful but they are difficult to read and write due to the presence of so many ~:_ directives. A special form of the ~!...~. directive is provided which makes it easier to specify such FORMAT control strings. If the atsign flag is used with the directive (i.e., ~@; or ~@.) that marks the end of the body in a ~!...~. directive then the body is modified by automatically inserting ~:_ directives after each chunk of which space (except for white space after a ~<newline> directive). The example below shows the use of this feature. It is exactly equivalent to the example above.

```
(FERROR "~:@!;~@;Bound variable ~S is not a ~
                 symbol in ~S.~@."
        2 '(LET ((2 X-VALUE)) (+ X-VALUE X-VALUE)))
```

Indentation

As mentioned above, the second and subsequent output lines corresponding to a logical block are indented so that they line up vertically under the column position of the first character in the block after the prefix. The FORMAT directive ~I makes it possible to specify a different indentation. The directive ~*n*I specifies that the indentation should be set to the column position of the first character after the prefix plus *n*. The directive ~*n*:I specifies that the indentation should be set to the column position in the output of the ~*n*:I directive itself, plus *n*. If omitted, the parameter *n* defaults to zero. The parameter can be negative, but the indentation cannot be set at less than the position of the first character after the prefix. If a ~I directive is encountered outside of a logical block, it is ignored. As an example of using ~I, consider the following.

```
(FORMAT T "~1:I~S ~:I~S ~:_~S ~1I~_~S~." '(DEFUN PRODUCT (X Y) (* X Y)))
```

If the line width available is 25, multi-line mode will be used and a newline will be inserted at the multi-line mode conditional newline as shown below. The directive ~1I specifies that the statement in the body of the DEFUN should be printed at a relative indentation of 1 in the logical block. Note that an open parenthesis is used as a prefix in the logical block, and therefore the total indentation is 2.

```
(DEFUN PRODUCT (X Y)
  (* X Y))
```

If the line width is less than 21, a newline will also be inserted at the if-needed conditional newline before the argument list as shown below. The directive ~:I before the function name causes the argument list to be lined up under the name. Notice that the column position corresponding to the ~:I directive is determined dynamically as the output is produced.

```
(DEFUN PRODUCT
       (X Y)
  (* X Y))
```

Changes in relative indentation caused by a ~I directive do not take effect until after the next newline is printed. As a result, it is important that the ~1I directive in the call on FORMAT above precede the ~_ directive after the argument list. In addition, a ~I directive only affects the indentation in the immediately containing logical block.

Miser Mode

A fundamental problem with pretty printing arises when printing very large structures. As the pretty printer is forced to print more and more deeply nested logical blocks in multi-line mode, the indentation gets greater and greater. This causes the line width available for printing to get smaller and smaller until it is no longer possible to print substructures in the space allowed.

An approach to dealing with this problem which has been used at least since the original Goldstein pretty printer [1] is to introduce a special compact kind of multi-line mode (called *miser* mode) and to use this mode once the line width begins to get small. The key idea behind miser mode is that by trading off readability for compactness it reduces the width necessary for printing a logical block, and even more importantly, it reduces the indentation required for printing the output lines in the block.

In the PP system, the use of miser mode is controlled by the variable PP:*PRINT-MISER-WIDTH*. Miser mode is used instead of ordinary multi-line mode whenever the line width available is less than the value of PP:*PRINT-MISER-WIDTH*. Single-line mode is always used when a logical block can fit in the line width available. If PP:*PRINT-MISER-WIDTH* is given the value NIL then miser mode will never be used.

Miser mode has two major effects on the way a logical block is printed. First, newlines are inserted at every if-needed conditional newline as well as at every multi-line mode conditional newline. Second, all ~I directives are ignored, thereby forcing the lines corresponding to the logical block to line up under the first character in the block after the prefix. Consider again the example (reproduced below) used in the last subsection.

```
(FORMAT T "~1:I~S ~:I~S ~:_~S ~1I~_~S~." '(DEFUN PRODUCT (X Y) (* X Y)))
```

If the line width available were 25 and less than PP:*PRINT-MISER-WIDTH*, the output would be printed as shown below. (For simplicity, the examples in this paper always assume that miser mode printing is not triggered, unless it is explicitly stated otherwise.)

```
(DEFUN PRODUCT
 (X Y)
 (* X Y))
```

The variable PP:*PRINT-MISER-WIDTH* has a default value of 40. A constant value irrespective of line length is used because the point at which miser mode should be triggered does not depend on the line length, but rather on how long the logical block sections between conditional newline positions are in the output being printed. When pretty printing programs the distance between conditional newlines depends on the lengths of the symbols in the program. Experience suggests that PP:*PRINT-MISER-WIDTH* should be set at from two to four times the length of the typical symbol being printed.

Even in miser mode, the pretty printing algorithm used by PP is not guaranteed to succeed in keeping its output within the line width available. The pretty printing algorithm never inserts newlines other than at conditional newline positions. As a result, a given output requires a certain minimum amount of line width in order to print it (e.g., a width of 14 in the example above). If the amount of line width available is less than this amount, the pretty printing algorithm simply prints characters beyond the end of the line.

The GPRINT system supported an additional mechanism for dealing with the problem of running out of line width when pretty printing deeply nested structures. When indentation reduced the line width to a small percentage of its initial value, major program structures (such as PROGs and LETs) were shifted to the left by reducing the indentation. This violated standard Lisp pretty printing style, but significantly increased the line width available for printing. Experience showed that though this was very useful in some situations, it was, in general, more confusing than helpful. As a result, this feature was not included in the PP system.

Miser Mode Conditional Newlines

In order to give the user greater control over how a logical block will be printed in miser mode, PP provides a third kind of conditional newline called a *miser mode* conditional newline. A newline is inserted at a miser mode conditional newline if and only if the immediately containing logical block is being printed in miser mode. A miser mode conditional newline can be specified by using the FORMAT directive ~@_. If a ~@_ directive is encountered outside of a logical block, it is ignored. The example below is the same as the one in the last subsection except that a miser mode newline directive has been placed before the function name.

```
(FORMAT T "~1:I~S ~:I~@_~S ~:_~S ~2I~_~S~." '(DEFUN PRODUCT (X Y) (* X Y)))
```

In this case, if the line width available were 25 and less than PP:*PRINT-MISER-WIDTH*, the output would be printed as shown below, further reducing the line width needed.

```
(DEFUN
 PRODUCT
 (X Y)
 (* X Y))
```

Pretty Printing as Selection

Stepping back a moment, it is useful to reflect on how the FORMAT directives above interact in order to support pretty printing. The ~I...~. directives in a FORMAT control string divide the output up into a hierarchy of logical blocks. The ~_ and ~I directives simultaneously specify three ways (single-line mode, multi-line mode, and miser mode) in which each logical block can be printed. The job of the pretty printer boils down to selecting (based on the values of *PRINT-PRETTY* and PP:*PRINT-MISER-WIDTH*, on the length of the block, and on the line width available for printing) which of the three modes each logical block will be printed in.

The FORMAT directives have been designed so that it is relatively easy to specify three different ways to print a logical block in a single FORMAT control string. None of the special FORMAT directives generate any output other than prefixes, suffixes, newlines, and indentation. The characters to be output in each logical block section are determined solely by the rest of the FORMAT control string and are exactly the same in each of the three modes. The ~_ and ~I directives specify how the logical block sections are to be arranged in multi-line mode and miser mode. A certain amount of care has to be taken to make sure that output is aesthetic in both of these modes.

There are many other kinds of pretty printing directives which could have been supported -- for example, sections of text which are output only in multi-line mode. The PP system supports only the limited set of directives above because, experience has shown them to be a good compromise between the requirements of expressive power, easy understandability, and efficiency.

Interaction With the Standard Format Directives

All of the standard FORMAT directives can be used in conjunction with the new directives described above. It should be noted that the standard FORMAT directives ~A and ~S are sensitive to the variable *PRINT-PRETTY* and will therefore do pretty printing when used inside ~:I...~..

A few of the standard FORMAT directives have dynamic formatting capabilities. For example, the directive ~<...~:;...~> conditionally inserts a newline with a prefix, and the directive ~→...~← uses indentation when printing. These directives are largely rendered obsolete by the new directives introduced above.

Three of the standard FORMAT directives have been modified so that they will fit in better with pretty printing. These are discussed in the following two subsections.

Tabbing Within a Logical Block

If encountered in a logical block, the standard FORMAT directive ~T is altered in its behavior. In this situation, ~T acts the same as normal except that it spaces relative to the beginning of the immediately containing logical block section, rather than with respect to the beginning of the line as a whole. In addition, while the atsign flag is supported, the colon flag for ~T is not supported in a logical block.

As an example of using ~T in a logical block, consider the following call on FORMAT. Each street name is followed by a space and a ~8T which ensures that the total width taken up will be 8. If-needed conditional newlines are used to put as many streets as possible on each line.

```
(FORMAT T "Roads ~:I~@{~S~^ ~8T~:_~}~." '(ELM MAIN MAPLE CENTER))
```

If the line width is 25, this will print in two columns as shown below.

```
Roads ELM     MAIN
      MAPLE   CENTER
```

The change in meaning of ~T in a logical block has two advantages. First, because it operates within a logical block section, it is independent of whatever indentation is in effect. In the example above, a column spacing of 8 is used, but the entire table is shifted over 6 columns due to the indentation of the logical block as a whole. Second, at the time of this writing, an unfortunate aspect of the standard ~T directive is that it only does tabbing when the output stream supports the messages :READ-CURSORPOS and :SET-CURSORPOS and yet many streams do not support these messages. The changed meaning of ~T gets around this problem because it can be handled internally by the pretty printing algorithm without using these messages.

Unconditional Newlines in a Logical Block

If encountered in a logical block when *PRINT-PRETTY* is T, the standard FORMAT directive ~% (unconditional newline) has its meaning altered slightly. In addition to unconditionally inserting newlines in thc normal way, indentation is used (and any per-line prefixes are inscrted) just as with the ~_ dircctive.

The pretty printing algorithm is also somewhat modified in order to take account of ~%. The inclusion of a ~% directive in a logical block prevents the block from bcing printcd on a single linc. In order to deal with this aesthetically when pretty printing, a logical block is automatically printed in multi-line mode whenever it contains a ~% directive.

As an example of the above, consider the following call on FORMAT.

```
(FORMAT T "Roads ~:I;~@;ELM ~%MAIN ~_MAPLE~.")
```

The output below will be produced no matter how much line width is available for printing.

```
Roads ;ELM
      ;MAIN
      ;MAPLE
```

Note that ~% forces the inscrtion of a newline even when not prctty printing. In this case a ~% directive will not cause a logical block to be printed in multi-linc modc. However, indcntation and any per-line prefix will still be printcd aftcr the newline caused by the ~%. As a result, thc following output would have been produced if the colon flag had not bcen used with the ~I...~. directive above.

```
Roads ;ELM
      ;MAIN MAPLE
```

If encountered in a logical block when *PRINT-PRETTY* is T, the standard FORMAT directive ~& (fresh line) has its meaning altered in the same way as ~%. If it causes the inscrtion of a newlinc, then indentation is used and the containing logical blocks are printed in multi-line mode. In addition, thc computation of when to insert a newline at a ~& directive is altered so that it operates with respect to the immediately containing logical block. A newline is inserted only if some characters other than indentation and the prefix have been output in the block since the last newline. This is illustrated by the following example which produces the same output as the last example.

```
(FORMAT T "Roads ~:I;~@;~&ELM ~&MAIN ~_MAPLE~.")
```

Checking for Malformed Argument Lists

A problem with FORMAT arises when a ~{...~} directive is used to iterate over a list argument which has a non-NIL atomic CDR. This is illustrated in the example below which generates an error trying to take the CAR of 2.

```
(FORMAT T "(~{~S~^ ~S~})" '(1 . 2))
```

The problem is that, using the standard FORMAT directives, there is no way to detect when a non-NIL atomic CDR is encountered. The ~^ directive tests only whether the argument list has been reduced to NIL. In order to deal with this problem, ~^ has its meaning altered slightly when it is encountered in a logical block. In addition to testing whether the argument list is exhausted, it checks to see if the argument list has been reduced to a non-NIL atom. If checking reveals that the argument list has become a non-NIL atom then " . " is printed followed by the atomic value and the processing of the body of the immediately enclosing logical block is terminated. Any suffix which has been specified for the logical block is printed. The effects of this check are illustrated in the example below.

```
(FORMAT T "~1I~S~^ ~S~." '(1 . 2))
```

The call on FORMAT produces the following output.

```
(1 . 2)
```

Note that the check for a non-NIL atomic argument list is performed even though pretty printing is not triggered in this example.

The check for a non-NIL atomic argument list is particularly useful during debugging because it makes it possible for a FORMAT control string to execute without error and produce informative output even when its arguments are malformed.

User-Defined Format Directives

The directives ~A, ~S, and ~W all call the standard output functions in order to print the corresponding FORMAT argument. The standard output functions decide how to print an object based on the characteristics of the object. Occasionally it is important to be able to bypass this decision and specify a non-standard way to print something. PP provides a mechanism for allowing the user to define new FORMAT directives which specify a special function to be used for printing the corresponding FORMAT argument.

PP:DEFINE-FORMAT-DIRECTIVE *name* (*arg stream colon atsign* &rest *parameters*) &body *forms*

This macro defines a FORMAT directive which prints a FORMAT argument. The directive can be used by putting ~*package:name*\ in a FORMAT control string. (When Common Lisp compatibility is introduced, such directives will be used by putting ~/*package:name*/ in the FORMAT control string.) If the package prefix is omitted it defaults to the package FORMAT rather than to the current package.

When used, a user-defined FORMAT directive is called with the arguments shown. The first argument *arg* is the argument to FORMAT which is to be printed by the directive. The second argument *stream* is the stream to print *arg* on. The next two arguments *colon* and *atsign* are non-NIL if and only if the colon and/or atsign flags (respectively) were used with the directive. The remaining arguments are the parameters specified for the directive in the FORMAT control string (if any). The *forms* are evaluated to print *arg* on *stream*.

The definition of a FORMAT directive is illustrated below. The directive takes a symbol as its argument and prints it out followed by its value.

```
(PP:DEFINE-FORMAT-DIRECTIVE SHOW-VALUE (SYMBOL STREAM &REST IGNORE)
  (FORMAT STREAM "~S = ~S" SYMBOL (SYMEVAL SYMBOL)))

(LET ((*ROADS* '(ELM MAIN)))
  (FORMAT T "~\USER:SHOW-VALUE\" '*ROADS*))
```

The call on FORMAT which uses the user defined FORMAT directive SHOW-VALUE produces the following output.

```
*ROADS* = (ELM MAIN)
```

The approach presented above for supporting the user specification of a printing function to use should be contrasted with the standard FORMAT directive ~Q. This directive takes an argument which is a function to use to print a second argument. The problem with this approach is that it requires the function to be embedded in the argument list. If the FORMAT directive ~{...~} is being used then this requires that the function be embedded inside of a list which is one of the arguments. This is an inconvenient requirement.

The macro PP:DEFINE-FORMAT-DIRECTIVE is built on top of the standard macro FORMAT:DEFFORMAT and shares many of the latter macro's idiosyncrasies -- e.g., the fact that when a user defined FORMAT directive is used the package prefix defaults to the FORMAT package. The macro PP:DEFINE-FORMAT-DIRECTIVE has two advantages over the macro FORMAT:DEFFORMAT. First, though it is less powerful, it is simpler to use. Second, the function implementing the FORMAT directive defined takes all of its inputs as functional arguments, rather than as free variable inputs.

Special Format Directives For Lists

PP supplies three special FORMAT directives for printing Lisp lists. The first of these is called ~\FILL\ and prints out the elements of a list with as many elements as possible on each line. The definition of this directive is shown below. In analogy with ~!...~., ~\FILL\ takes a numerical parameter (which defaults to 0) that specifies whether parenthesis should be printed around the list elements.

```
(PP:DEFINE-FORMAT-DIRECTIVE FORMAT:FILL (LIST STREAM IGNORE IGNORE
                                         &OPTIONAL (PARENS? 0))
  (PP:FAST-FORMAT STREAM "~V!~@{~W~^ ~:_~}~." PARENS? LIST))
```

The standard FORMAT directives ~@{...~} (iterate over arguments), and ~^ (terminate iteration) are used in order to iterate over the list argument. A ~W is used to print each element of the list. A space, and an if-needed conditional newline are printed after each element except for the last one. The use of ~\FILL\ is illustrated below.

```
(FORMAT T "~:@!~1\FILL\~." '(ONE TWO THREE FOUR FIVE SIX . SEVEN))
```

With a line length of 20 the call on FORMAT above produces the following output.

```
(ONE TWO THREE FOUR
 FIVE SIX . SEVEN)
```

Four things should be noticed about the style in which the directive ~\FILL\ is written. First, it does not force pretty printing to occur. It simply specifies how pretty printing should be carried out when *PRINT-PRETTY* is T. Second, ~W is used to print out the elements of the list so that the printing of escape characters will neither be forced nor prevented. Third, the function PP:FAST-FORMAT is used on the theory that efficiency is important since ~\FILL\ is called many times in a wide variety of contexts. Fourth, due to the fact that ~^ checks for the presence of a non-NIL atomic CDR, ~\FILL\ produces reasonable output even when it is given a malformed list as its argument.

The second special FORMAT directive for lists is called ~\LINEAR\ and prints a list with one element on each line when the list cannot be printed on a single line. The third is called ~\TABULAR\ and prints a list in a tabular form. Both of these directives take a parameter which, if specified as 1, causes parentheses to be printed around the list elements. The directive ~\TABULAR\ takes a second a parameter (default 16) which specifies the column spacing to use. These two directives are demonstrated in the call on FORMAT below.

```
(LET ((X '(ELM MAIN MAPLE CENTER)))
  (FORMAT T "~:@!~1\LINEAR\~%~,6\TABULAR\~." X X))
```

If the line length is 30 then this will produce the following output.

```
(ELM MAIN MAPLE CENTER)
ELM    MAIN   MAPLE  CENTER
```

If the line width is 15 then this will produce the following output.

```
(ELM
 MAIN
 MAPLE
 CENTER)
ELM    MAIN
MAPLE  CENTER
```

Controlling the Margins

In order to work properly, the pretty printing algorithm used by PP needs to know the line length available for output. In general, this is determined by querying the output stream when the outermost logical block is entered. However, if the output stream does not support the :SIZE-IN-CHARACTERS message then a line width of 95 is assumed.

The variable PP:*PRINT-RIGHT-MARGIN* (which has a default value of NIL) can be used to control the right margin position. If this variable is non-NIL then it will override the stream and be used as the line width.

The user can exercise control over the left margin by simply creating a logical block which begins at the desired indentation. The pretty printing algorithm will then continue this indentation.

In order to know how much indentation to use inside of a logical block, the pretty printing algorithm needs to know the column position where the logical block begins. For all but the outermost logical block, the column position of the start of a block is determined internally by the pretty printing algorithm relative to the starting position of the outermost block. In order to determine the starting position of the outermost logical block, the pretty printing algorithm sends the output stream a :READ-CURSORPOS message. If the output stream does not support this message, then the pretty printing algorithm assumes that the outermost logical block begins in column zero.

As an example of the above points, consider the following function which pretty prints an object between two specified margins.

```
(DEFUN PPRINT-BETWEEN-MARGINS (OBJECT STREAM LEFT RIGHT)
  (LET ((PP:*PRINT-RIGHT-MARGIN* RIGHT))
    (FORMAT STREAM "~%~@I~VT~@I~:W~.~." LEFT OBJECT)))
```

The function sets the right margin by binding the variable PP:*PRINT-RIGHT-MARGIN*. It then sets the left margin by going to a new line (~%), tabbing over to the column specified by LEFT (~VT), and starting a logical block. The object to be printed is then pretty printed inside the block. An outermost logical block is started in column zero in order to insure that the pretty printing algorithm will determine the correct starting column and that the ~VT will work even if STREAM does not support the :READ-CURSORPOS message.

IV - Print-Self Functions

A key feature of the standard output functions is the idea of print-self methods specifying how to print flavor instances and named data structures implemented as arrays. The discussion below assumes a basic understanding of print-self methods. The Lisp Machine documentation [4] describes them in detail. (Though Common Lisp does not have print-self methods for flavors, it does support the concept of print-self methods (called :PRINT-FUNCTIONs) for structures.)

Dispatching to print-self methods is supported during pretty printing in exactly the same way as it is supported during ordinary printing. Consider the following print-self method which prints out the instance variables of a flavor called FAMILY.

```
(DEFFLAVOR FAMILY (MOTHER CHILDREN) ()
  (:INITABLE-INSTANCE-VARIABLES MOTHER CHILDREN))

(DEFMETHOD (FAMILY :PRINT-SELF) (STREAM &REST IGNORE)
  (PP:FAST-FORMAT STREAM "~@!#<~;FAMILY - ~:IMOTHER ~W ~_CHILDREN ~1\FILL\~;>~."
                  MOTHER CHILDREN))

(SETQ F (MAKE-INSTANCE 'FAMILY
                       ':MOTHER "Lucy"
                       ':CHILDREN '("Mark" "Barbara" "Danny" "Sally")))
```

Evaluating (PPRINT F) when the line width is 45 produces the following output. In the course of pretty printing, the print-self method is called to print the instance.

```
#<FAMILY - MOTHER "Lucy"
           CHILDREN ("Mark" "Barbara"
                     "Danny" "Sally")>
```

Evaluating (PRINT F) with *PRINT-PRETTY* bound to NIL would produce the output below no matter how little line width were available for printing because nothing in the the print-self method above, or in the directive ~\FILL\ forces pretty printing to occur. An important aspect of the PP system is that it allows you to write print-self methods which specify how something should be pretty printed without forcing pretty printing to occur.

```
#<FAMILY - MOTHER "Lucy" CHILDREN ("Mark" "Barbara" "Danny" "Sally")>
```

Evaluating (PRINC F) would produce the following output, since nothing in the print-self method, or in the directive ~\FILL\ forces the printing of escape characters.

```
#<FAMILY - MOTHER Lucy CHILDREN (Mark Barbara Danny Sally)>
```

List Print-Self Functions

Print-self methods cannot be defined for lists. This is unfortunate since lists are used to represent both programs and a wide variety of data structures. PP supports a concept (similar to print-self methods) for lists which makes it possible to specify how to pretty print particular program constructs, and particular kinds of list data structures. PP looks at the CAR of every list to be pretty printed and determines whether a special list print-self function has been defined for lists which begin with this CAR. It is important to note that dispatching to list print-self functions is only performed during pretty printing. When *PRINT-PRETTY* is NIL all lists are printed in the standard way.

PP:DEFINE-LIST-PRINT-FUNCTION *symbol* (*list stream*) &body *forms*

This macro defines a list print-self function for lists whose CAR is *symbol.* List print-self functions are called with the arguments shown. The *forms* are evaluated to print *list* on *stream.*

When a list print-self function is defined for a symbol it replaces any list print-self function which was previously defined for that symbol. In order to remove a previously defined list print-self function without defining a new one, you can use the function (PP:UNDEFINE-LIST-PRINT-FUNCTION *symbol*).

As an example of using PP:DEFINE-LIST-PRINT-FUNCTION consider the following definition of how to print CONDs. The FORMAT string prints the first element of the list followed by a space. It then sets the indentation at the character position corresponding to this point and specifies a miser mode conditional newline. It then uses ~@{...~} to iterate over the list of COND clauses using the directive ~\LINEAR\ to print each clause.

```
(PP:DEFINE-LIST-PRINT-FUNCTION COND (LIST STREAM)
  (PP:FAST-FORMAT STREAM "~1!~W~^ ~:I~@_~@{~1\LINEAR\~^ ~_~}~." LIST))
```

The above list print-self function causes CONDs to be printed in the standard style. If a COND cannot be printed on a single line then each clause is printed on a separate line. If a clause cannot be printed on a single line, then each element of the clause is printed on a separate line. This is illustrated in the output below.

```
(COND ((TEST X) (DO-THIS X))
      (COND
       (SETQ X (CAR X))
       X))
```

A basic concept behind list print-self functions is that the *type* of a list can be determined by looking at its CAR. This is true much of the time when looking at program constructs and named data structures implemented as lists. However, it is not true all the time. This can cause the pretty printer to make mistakes. For example, if a list begins with the symbol COND then it will in general always be printed with the list print-self function above even if it is not actually a COND form (e.g., even if it is just an arbitrary piece of data such a list of function names).

An important aspect of the user-defined FORMAT directives is that they can be used to prevent the erroneous application of list print-self functions. When a user-defined FORMAT directive is used to print a list, then the standard output functions are not called and therefore, dispatching to list print-self functions does not occur. The print-self function for COND above uses the directive ~\LINEAR\ to print each COND clause. As a result, even though the first element of the second clause is the symbol COND the second clause is printed in linear style, and not in COND style. In general, user-defined FORMAT directives should be used instead of ~A, ~S, or ~W whenever the way an object is to be printed is controlled by the context of its use rather than by its intrinsic properties.

Predefined List Print-Self Functions

In order to support traditional Lisp pretty printing style, the PP system provides predefined list print-self functions for all of the standard Lisp program constructs. One example of these is the print-self function for COND shown in the last subsection. Another example is the list print-self function for QUOTE shown below. The user can change the way any given kind of list is printed by defining a new list print-self function for it.

```
(PP:DEFINE-LIST-PRINT-FUNCTION QUOTE (LIST STREAM)
  (COND ((AND (LISTP (CDR LIST)) (NULL (CDDR LIST)))
         (PP:FAST-FORMAT STREAM "'~W" (CADR LIST)))
        (T (PP:FAST-FORMAT STREAM "~1\FILL\" LIST))))
```

The output below illustrates the way quoted objects are printed.

```
'(COND ((NULL X) 'NULL))
```

There are several interesting things to note about the way the print-self function for QUOTE is written. First, all of the predefined list print-self functions use PP:FAST-FORMAT in the interest of efficiency. Second, the print-self function checks to see that the LIST is well formed in order to guarantee that it will be printed in such a way that it can be read back in. Third, the list print-self function uses the full power of the pretty printer to print the quoted list itself so that the output will be aesthetic when the quoted list is a program construct as in the example.

During pretty printing, if a list has no list print-self function specified for it then the list is printed in one of three ways. If the CAR of the list is the name of a defined function (i.e., is FBOUNDP), then the list is printed as a function call. In order to do this the variable ZWEI:*LISP-INDENT-OFFSET-ALIST* is consulted to determine what indentation pattern to use. As a result of this, the pretty printer automatically knows how to indent correctly everything that Zwei knows how to indent correctly. This includes understanding macros that are defined using the keyword &BODY. If a list is a literal LAMBDA combination (i.e., if the CAR of the list is a list whose CAR is the symbol LAMBDA) then the list is printed using ~\LINEAR\. In all other cases, the list is printed using ~\FILL\.

When *PRINT-PRETTY* is NIL, then every list is printed in the traditional way with each sublist grouped together as a logical block.

Generalized Print-Self Functions

PP supports a generalized kind of print-self function which makes it possible to specify how to pretty print objects other than flavor instances and lists with CARs which are symbols.

PP:DEFINE-GENERALIZED-PRINT-FUNCTION *name* (*object stream*) *predicate* **&body** *forms*

This macro defines a generalized print-self function named *name*. Generalized print-self functions are called with the arguments shown. The *predicate* is used to determine whether the generalized print-self function is applicable to a given object. When the function is applicable, the *forms* are evaluated to print *object* on *stream*.

Unlike list print-self functions, defining a generalized print-self function does not cause the pretty printer to use it. It must be explicitly enabled before it will be used.

PP:GENERALIZED-PRINT-FUNCTION-ENABLED-P *name*

Determines whether the named generalized print-self function is enabled. SETF can be used to enable or disable a given generalized print-self function.

PP:WITH-GENERALIZED-PRINT-FUNCTION *name* **&body** *forms*

This macro evaluates the *forms* with the specified generalized print-self function enabled.

Whenever any object is to be pretty printed, every enabled generalized print-self function is tested to see if it is applicable. If one is applicable then it is used to pretty print the object. (Note that when a user-defined FORMAT directive is used to print an object, then the standard output functions are not called and therefore, dispatching to generalized print-self functions does not occur.) If more than one generalized print-self function is applicable, then the most recently enabled function will be used. It is important to note that dispatching to generalized print-self functions is only performed during pretty printing.

The example generalized print-self function below causes NIL to print out as "()".

```
(PP:DEFINE-GENERALIZED-PRINT-FUNCTION NIL-AS-EMPTY-LIST (OBJ STREAM)
    (NULL OBJ)
  (PRINC "()" STREAM))
```

The use of this generalized print-self function is illustrated below.

```
(PPRINT NIL)
(PP:WITH-GENERALIZED-PRINT-FUNCTION NIL-AS-EMPTY-LIST
  (PRINT NIL)
  (PPRINT NIL))
(SETF (PP:GENERALIZED-PRINT-FUNCTION-ENABLED-P 'NIL-AS-EMPTY-LIST) T)
(PPRINT NIL)
```

The above forms produce the following output. The first call on PPRINT does not trigger the generalized print-self function because it is not enabled. The call on PRINT does not trigger it because pretty printing is not enabled. The second call on PPRINT triggers the generalized print-self function because the function is locally enabled. The third call on PPRINT triggers the generalized print-self function because the function has been globally enabled.

```
NIL
NIL
()
()
```

Tailoring the Pretty Printer

An important aspect of generalized print-self functions is that they serve as a general mechanism for tailoring the pretty printer. For example, the pretty printer prints lists with non-atomic CARs using the directive ~\FILL\. Suppose that a user wanted to have lists with non-atomic CARs printed using ~\LINEAR\ instead. This change could be made by enabling the following generalized print-self method.

```
(PP:DEFINE-GENERALIZED-PRINT-FUNCTION LINEAR-LISTS (OBJ STREAM)
    (AND (LISTP OBJ) (NOT (ATOM (CAR OBJ))))
  (PP:FAST-FORMAT STREAM "~1\LINEAR\" OBJ))
```

As a more interesting example, suppose that the user wanted lists with non-atomic CARs to be printed using ~\FILL\ in general, but wanted ~\LINEAR\ to be used when a list with a non-atomic CAR occurred inside a quoted structure. This change could be made be using the following list print-self function for QUOTE. The list print-self function makes the change by selectively enabling the generalized print-self function LINEAR-LISTS only when printing a quoted structure.

```
(PP:DEFINE-LIST-PRINT-FUNCTION QUOTE (LIST STREAM)
    (COND ((AND (LISTP (CDR LIST)) (NULL (CDDR LIST)))
           (PP:WITH-GENERALIZED-PRINT-FUNCTION LINEAR-LISTS
             (PP:FAST-FORMAT STREAM "'~W" (CADR LIST))))
          (T (PP:FAST-FORMAT STREAM "~1\FILL\" LIST))))
```

Generalized print-self functions can be very useful. However, since every enabled generalized print-self function must be tested against every object to be pretty printed, the generalized print-self mechanism is rather inefficient. It is suggested that generalized print-self functions only be locally enabled (using PP:WITH-GENERALIZED-PRINT-FUNCTION) where specifically needed. In general, globally activating a generalized print-self function is only reasonable during debugging.

V - Abbreviation

This section describes the way the PP system handles the abbreviation of output. Abbreviation of output based on the length and depth of nested structures is supported both during ordinary printing and during pretty printing. The way they are supported is changed so that they are automatically applied to every kind of structure without the program writer having to take any explicit action when writing a print-self method. In addition, a new kind of abbreviation is introduced which can be used to limit the total number of lines printed.

Length Abbreviation

Length abbreviation is controlled by the variable PRINLENGTH. (When Common Lisp compatibility is introduced this variable will be called *PRINT-LENGTH*.) The standard output functions for both the Lisp Machine and Common Lisp provide no automatic support for length abbreviation except that the internal function for printing lists performs abbreviation when it is appropriate. If a user wants to support length abbreviation in a print-self method, then he has to consult the value of PRINLENGTH and print an abbreviation marker when appropriate. However, due to the difficulty of actually doing this, print-self methods seldom support length abbreviation. Therefore, as a practical matter, length abbreviation is only supported for lists.

The PP system handles length abbreviation in a different way. Length abbreviation is handled automatically based on the number of sections that are printed inside of a logical block. As a result, the writer of a print-self method need not take any action at all, other than specifying logical blocks, in order to support length abbreviation.

Each time a ~_, ~%, or ~& directive ending a logical block section is encountered the PP system checks how many sections have been printed inside of the current logical block. If PRINLENGTH sections have been printed, then "..." is printed and the processing of the body of the immediately enclosing logical block is terminated. Any suffix which has been specified for the logical block is printed. As an example of how length abbreviation works, consider the print-self method (reproduced below) for the flavor called FAMILY.

```
(PP:DEFINE-FORMAT-DIRECTIVE FORMAT:FILL (LIST STREAM IGNORE IGNORE
                                         &OPTIONAL (PARENS? 0))
  (PP:FAST-FORMAT STREAM "~V!~@{~W~^ ~:_~}~." PARENS? LIST))

(DEFMETHOD (FAMILY :PRINT-SELF) (STREAM &REST IGNORE)
  (PP:FAST-FORMAT STREAM "~@!#<~;FAMILY - ~:IMOTHER ~W ~_CHILDREN ~1\FILL\~;>~."
                  MOTHER CHILDREN))
```

Executing (PRINT F) with PRINLENGTH set to 1 produces the following.

```
#<FAMILY - MOTHER "Lucy" ...>
```

Executing (PRINT F) with PRINLENGTH set to 2 produces the following.

```
#<FAMILY - MOTHER "Lucy" CHILDREN ("Mark" "Barbara" ...)>
```

Length abbreviation is automatically supported everywhere logical blocks are used. This is the case even when *PRINT-PRETTY* is NIL. Length abbreviation can be prevented by binding PRINLENGTH to NIL (e.g., by using the colon flag with ~W).

Depth Abbreviation

Depth abbreviation is controlled by the variable PRINLEVEL. (When Common Lisp compatibility is introduced this variable will be called *PRINT-LEVEL*.) The standard output functions for both the Lisp Machine and Common Lisp handle depth abbreviation by passing around a parameter which specifies the printing depth at each level in the structure being printed. In order to fit in with this scheme, print-self methods are passed a printing depth argument and are expected to handle it appropriately -- e.g., to print out a depth abbreviation marker when necessary, and to pass along an incremented depth counter when printing the subparts of a structure. However, due to the difficulty of actually doing this, print-self methods typically ignore their printing depth argument. Therefore, as a practical matter, depth abbreviation is only supported by the standard printer for nested lists.

The PP system handles depth abbreviation in a different way. Rather than passing around a depth counter, depth abbreviation is handled based on the dynamic nesting of logical blocks in the output. As a result, the writer of a print-self method need not take any action at all, other than specifying logical blocks, in order to support depth abbreviation.

If a logical block is started at a dynamic nesting depth of PRINLEVEL in logical blocks, then ** is printed in the output, and execution goes to after the end of the logical block -- skipping the computation of what to print in the block. (When Common Lisp compatibility is introduced # will be printed in the output.) Note that the dynamic nesting depth of a logical block is not the static nesting depth of the corresponding ~!...~. directive in its FORMAT control string, but the total depth in the output. As an example of how depth abbreviation works, consider again the print-self method (reproduced below) for the flavor called FAMILY.

```
(PP:DEFINE-FORMAT-DIRECTIVE FORMAT:FILL (LIST STREAM IGNORE IGNORE
                                         &OPTIONAL (PARENS? 0))
  (PP:FAST-FORMAT STREAM "~V!~@{~W~^ ~:_~}~." PARENS? LIST))

(DEFMETHOD (FAMILY :PRINT-SELF) (STREAM &REST IGNORE)
  (PP:FAST-FORMAT STREAM "~@!#<~;FAMILY - ~:IMOTHER ~W ~_CHILDREN ~1\FILL\~;>~."
                  MOTHER CHILDREN))
```

Executing (PRINT F) with PRINLEVEL set to 1 produces the following. The list of children is abbreviated to ** because, when ~\FILL\ is called to print the list it attempts to start a logical block at a dynamic nesting depth of 1. The string "Lucy" is not abbreviated because no logical block is created when it is printed.

```
#<FAMILY - MOTHER "Lucy" CHILDREN **>
```

Suppose that the list of children were changed to (("Mark") "Barbara"). Executing (PRINT F) with PRINLEVEL set to 2 would produce the following.

```
#<FAMILY - MOTHER "Lucy" CHILDREN (** "Barbara")>
```

Depth abbreviation is automatically supported everywhere logical blocks are used. This is the case even when *PRINT-PRETTY* is NIL. Depth abbreviation can be prevented by binding PRINLEVEL to NIL (e.g., by using the colon flag with ~W).

For completeness, the PP output functions pass an appropriate depth counter value to print-self methods, however there is no reason for a print-self method to do anything with it. In the interest of simplicity, no depth counter argument is passed to list print-self functions, generalized print-self functions, or user-defined FORMAT directives.

Abbreviation Based on the Number of Lines Printed

In addition to the traditional abbreviation mechanisms, PP provides a new abbreviation mechanism which can be used to limit the number of lines printed. When non-NIL, the variable PP:*PRINT-LINES* controls the number of lines which can be printed by an outermost logical block. If an attempt is made to print more than PP:*PRINT-LINES* lines in an outermost logical block, then " ---" is printed at the end of the last line and the outermost logical block is immediately exited. In order to insure that printing can never go beyond the end of the last line, any pending suffixes are not printed.

As an example of abbreviation based on the number of lines printed, consider the following.

```
(LET ((PP:*PRINT-LINES* 3))
  (PPRINT '(SETQ A 1 B 2 C 3 D 4)))
```

This call on PPRINT produces the output below.

```
(SETQ A 1
      B 2
      C 3 ---
```

Abbreviation based on the number of lines printed is automatically supported everywhere logical blocks are used. This is the case even when *PRINT-PRETTY* is NIL. (However, it should be noted that ordinary printing usually creates only a single line of output -- which may be very long.) This abbreviation can be prevented by binding *PRINT-LINES* to NIL, which is its default value (e.g., by using the colon flag with ~W).

Extensive experience with the earlier GPRINT system has shown that abbreviation based on the number of lines printed can be much more useful than the traditional depth and length abbreviation mechanisms. This is particularly true when the user wants to limit output to a small space. In order to do this with depth and length abbreviation, the length, and more significantly the depth, have to be limited to very small values such as 3 or 4. This has the unfortunate effect of often producing output which consists almost totally of "**"s and "..."s grouped in parentheses. In contrast, limiting the total number of lines printed to 3 or 2 or even 1 produces a more legible output. Simply put, seeing the first few lines of output is usually more informative then seeing only the top level skeletal structure of the output.

Reprinting an Abbreviated Object

When an object is abbreviated due to *PRINT-LINES* while being printed by PP:INTERACTIVE-WRITE, the object is saved. A function is provided which can be used to reprint a saved object (or any other object) without any abbreviation. When a saved object is reprinted, an attempt is made to overwrite the old partial output with the full output in order to save screen space. Objects are saved on a per process basis so that an object abbreviated in one process will not get reprinted in another process.

PP:PP &optional *object stream*

This is the same as PPRINT except that it prints *object* without any abbreviation. The *object* defaults to the object last saved because it was abbreviated by PP:INTERACTIVE-WRITE due to *PRINT-LINES*.

Typing the Lisp Machine interrupt sequence <FUNCTION> <RESUME> evaluates the expression (PP:PP) in the current process.

VI - Conclusion

Reduced to its essence, PP embodies a few key ideas. These ideas transcend the system itself and could be used in almost any output system. In general, these ideas are relatively independent and a given output system could pick and choose among them. However, careful design is needed in order for all the ideas to fit together into a coherent whole.

The concept of logical blocks and conditional newlines provides a straightforward way for the writer of an output expression to deal reasonably aesthetically with two important kinds of variability. Variability in line length available for output, and variability in the sizes of the items being printed. The way PP adds logical blocks and conditional newlines into the standard Lisp FORMAT construct demonstrates that logical blocks and conditional newlines are orthogonal to most other output concepts and can be relatively easily integrated with them. The same would be true in virtually any programming environment.

PP extends the idea of having each object know how to print itself to lists. This provides a basis which makes it easy to implement traditional pretty printing of Lisp programs. It also makes it easy for a given user to control the way his programs pretty print. This idea could be used straightforwardly in any programming environment where a data structure representing program parse trees has been defined.

The closely related idea of generalized print-self functions provides a means for specifying how to print arbitrary objects. It also serves as a basis for tailoring the pretty printer in arbitrary ways.

An important philosophy underlying PP is that facilities should be provided so that a single output expression (e.g., FORMAT control string) can specify how to print something in each of the standard printing styles (i.e., with or without escape characters and with or without pretty printing) without having to force which style will be used. This gives the end user maximal flexibility in requesting different ways to print something. This philosophy is most evident in the fact that the FORMAT directive ~W does not force the printing of escape characters, and the directives ~!...~. and ~_ do not force pretty printing.

Depth and length abbreviation is applied to all output in a uniform way. This is done by giving depth and length abbreviation a simple definition in terms of logical blocks and logical block sections. This allows PP to support abbreviation for all output without the user having to write any explicit code to support it.

A new kind of abbreviation is introduced which can be used to limit the total number of lines printed. Experience has shown that in many situations this new abbreviation is more useful than depth and length abbreviation. This utility is enhanced by the availability of a mechanism for easily reprinting in full an object which was abbreviated based on the number of lines printed.

The idea of having separate interactive output control variables is introduced so that a user can specify what style should be used for interactive output without changing the default style which will be used for other output. This is particularly useful in conjunction with abbreviation which is typically only desirable during interactive output.

An important fact is the realization that pretty printing does not have to be slow. Because it is based on a fast linear algorithm, the pretty printer provided by PP is not significantly slower than the ordinary Lisp printer. Given that pretty printed output is a great deal more legible than ordinary output, there is little reason not to make pretty printing be the default (at least for interaction with the user).

Similarly, FORMAT does not have to be slower than any other output function -- it can in general be efficiently compiled. Most languages other than Lisp have always compiled their FORMAT-like statements.

Acknowledgements

A number of people have made important contributions to the development of the PP system. I would like to particularly thank K. Pitman and C. Rich as well as P. Anagnostopoulos, D. Chapman, and B. Morrison for making suggestions which significantly improved the system and this paper. I would also like to acknowledge the invaluable contribution of the many people involved in designing Lisp Machine Lisp in general and FORMAT in particular. There are few programming environments where it would be possible to write a system like PP as a simple user program integrated with the built in output facilities.

References

[1] Goldstein, I., "Pretty Printing, Converting List to Linear Structure", MIT/AIM-279, February 1973.

[2] Oppen, D., "Prettyprinting", ACM TOPLAS V2 #4, October 1980, pp. 465-483.

[3] Steele, G.L.Jr., "Common Lisp: the Language", Digital Press, Maynard MA, 1984.

[4] Symbolics, Lisp Machine documentation (release 5), Symbolics inc. Cambridge MA, 1984.

[5] Waters, R.C., "GPRINT: A LISP Pretty Printer Providing Extensive User Format-Control Mechanisms", MIT/AIM-611a, September 1982.

[6] Waters, R.C., "User Format Control in a Lisp Prettyprinter", ACM TOPLAS V5 #4 pp. 513-531, October 1983.

Appendix - Second Order Details

This section describes a number of details which are important in order to gain a full understanding of the pretty printing facilities provided by the PP system, but which are not important for the casual user.

The Pretty Printing Algorithm

Section III outlined the basic pretty printing algorithm used by PP. This subsection presents a number of special situations and boundary conditions which are not covered by the basic algorithm and describes how the algorithm has been extended to deal with them.

An area of complexity stems from the interaction of logical blocks and if-needed conditional newlines. Consider the call on FORMAT below

```
(FORMAT T "~1:I~1I1 ~:_2 ~:_3~. ~:_A ~:_B~.")
```

Using the pretty printing algorithm as defined above, this would create the following output given a line width of 8. The list (1 2 3) has to be printed in multi-line mode since it is too long to fit on one line. The element A is printed at the end of the second output line since there is room for it there.

```
((1 2
  3) A
 B)
```

The fact that A appears at the end of the second output line is judged by many people to violate a basic aesthetic criterion of Lisp pretty printing. In order to deal with this problem, the basic pretty printing algorithm has been altered so that a newline will be inserted at an if-needed conditional newline position whenever the logical block section preceding it is not printed on a single line. Applying this extended definition, the output is as follows.

```
((1 2
  3)
 A B)
```

Another area of complexity stems from the fact that a logical block is often followed by some additional characters which must be printed on the same line as the logical block. These characters can be literal characters such as the "." in the call on FORMAT below, or produced by FORMAT directives.

```
(FORMAT T "~:IRoads ~IELM ~_MAIN~..~.")
```

If the innermost logical block is printed on one line, then the "." will also have to be printed on that line. If there is only barely enough room for the logical block itself, then the "." will go over the end of the line. In order to deal with this problem, the decision of whether to use single-line mode or multi-line mode when printing a logical block (other than an outermost logical block) is not based solely on the length of the logical block itself, but rather on the length of the block plus the length of any characters which must be printed on the same line after it. For example, the call on FORMAT above produces multi-line mode output even when the line width available is equal to 14 -- i.e., just long enough for the block itself. When considering outermost logical blocks, characters that follow are not taken into account because the special printing supported by PP is no longer in effect after the end of an outermost logical block.

An analogous problem arises with regard to if-needed newlines. The last section of a logical block can be followed by characters which must be printed after it on the same line -- i.e., characters which follow the block as a whole. In order to allow for this, the decision of whether or not to insert a newline is based on the length of the logical block section itself plus the length of any characters which must be printed after it on the same

line unless they follow the outermost logical block.

A final area of complexity stems from the fact that there are many ways to force a newline in a FORMAT control string besides using the directives ~_, ~% and ~& -- e.g., putting a literal newline character in the control string or printing a string which contains a newline character. It is not completely obvious what to do in this situation. There are two very suggestive cases which unfortunately eontradict each other. First, suppose that a program which eontains a string constant whieh eontains a newline character is printed out and then read back in again. In order to insure that the result of the read will be EQUAL to the original program it is important that indentation not be inserted after the newline character in the string. On the other hand, suppose that the same program is being printed into a file by a FORMAT that specifies a prefix of ";;;" with the intention that the program will appear in the output as a comment. In order to insure that this comment will not interfere with subsequent reading from the file it is important that the prefix be printed after the newline in the string.

In order to try and satisfy the intent of both of the above cases, the following heuristie is applied. Indentation is used only if a newline is created with ~_, ~%, or ~&. However, per-line prefixes (and any indentation preeeding them) are always printed whenever a newline is created in the output no matter how the newline is specified. This heuristic is illustrated in the example below.

```
(LET ((X '((STRING-LENGTH "string on
two lines"))))
  (FORMAT T "~:!~W~.  ~%~:!;;;~@;~W~.  ~%For example: ~:!;~@;~W~." X X X))
```

This expression produces the following output.

```
(STRING-LENGTH "string on
two lines")
;;;(STRING-LENGTH "string on
;;;two lines")
For example: ;(STRING-LENGTH "string on
             ;two lines")
```

A small point centers around the fact that conditional newline specifications are typically preceded by some amount of blank space (see the examples above). This is done so that the sections of a logical block will be visually separated when the block is printed in single-line mode. Without anything more being said, this would lead to the printing of unnecessary blank spaces at the end of most lines in multi-line mode. In the interest of efficiency, the pretty printing algorithm suppresses the printing of blanks at the end of a line if (and only if) the newline ending the line is caused by a variant of the directive ~_.

Some Implementation Notes and Their Implications

The pretty printing algorithm is implemented in two parts. The first part is a modified version of the standard Lisp Machine function SI:PRINT-OBJECT which supports dispatching to print-self functions. The second part is a special kind of stream which is wrapped around the actual output stream. This intermediate stream buffers up the output and makes the dynamic pretty printing decisions. In order to install the pretty printing facilities in the PP system as a basie part of the standard Lisp Machine system, all that would be needed would be to replace SI:PRINT-OBJECT with its modified version and give every stream the capability to perform dynamic pretty printing decisions.

When pretty printing is initiated, an intermediate pretty printing stream is wrapped around the original output stream and then used as the destination of output. A beneficial effect of this approach is that it allows the user to use any kind of funetion to send output to the intermediate stream (e.g., in a print-self function). All such output is captured and processed. Output is only sent to the original stream after dynamie pretty

printing decisions have been made.

The intermediate stream has to buffer up output in order to make dynamic pretty printing decisions. The fundamental source of the efficiency of the pretty printing algorithm is that things are carefully designed so that the intermediate stream never has to buffer up more than one line width worth of output. The algorithm sends output to the underlying stream a line width worth at a time. The buffer is not guaranteed to be completely empty until the outermost logical block ends. Thus there is typically a delay between the time characters are sent to the intermediate stream and the time they appear on the underlying stream. This can be confusing if a process which is performing pretty printing is interrupted (e.g., during debugging).

Dynamic pretty printing decisions require a number of different calculations involving character positions and lengths of sections of text. All of these calculations are made while the output is buffered up in the intermediate stream. As a fundamental simplification, it is assumed that every character will use exactly one character position when actually output to the underlying stream. Only newlines are treated specially. This assumption can lead to problems in some situations. For example, it is inadvisable to use literal tab characters when pretty printing. It should be noted that (except for ~T) the standard FORMAT directives all make the same simplifying assumption.

Intermediate streams work by intercepting all of the basic output messages (i.e., :TYO, :STRING-OUT, :LINE-OUT, and :FRESH-LINE) being sent by output functions. In addition, they accept some special new messages which support logical blocks and conditional newlines.

In order to support all of the other messages (e.g., :READ-CURSORPOS) which may be supported by the underlying stream, an intermediate stream forwards any requests involving them directly to the underlying stream. One problem with this is that since some of these messages might perform output, the intermediate stream must empty out its buffer before forwarding a request. In order to do this, the intermediate stream must make decisions about any conditional newlines. When it does this, it pessimistically inserts newlines everywhere they cannot be immediately ruled out. As a result, the act of sending an intermediate stream a message (such as :READ-CURSORPOS) which it must send to the underlying stream can force the insertion of newlines which would not otherwise be inserted.

The user should avoid sending complex messages to an intermediate stream (e.g., in a print-self function). For example, an important reason why the behavior of the FORMAT directive ~T was altered in a logical block was so that it would no longer send :READ-CURSORPOS and :SET-CURSORPOS messages to the output stream.

Several aspects of the PP system (e.g., the abbreviation mechanisms) depend on knowing which logical block in the output is outermost. In order to determine this the following assumptions are made. Suppose that printing is in progress to stream A. If during this printing, printing is initiated on another stream (B), then it is assumed that the first block to occur during output to B is outermost. Further, suppose that during the output to B, output is again initiated to stream A. It is assumed that this is separate output to A, and that the first logical block to occur during this output to A will again be an outermost block.

The above assumptions are not necessarily valid. The user could be intending the output to streams A and B to operate as coroutines. In order for such coroutining output to work correctly in conjunction with the PP system, the two printing coroutines have to be in separate processes. It should be noted that the standard output functions implicitly make the same assumptions due to the way special variables are used to control many aspects of the printing process. However, the standard output functions do not rely on the assumptions so heavily.

Summary of Functions

PPRINT *object* **&optional** *stream*

This is the same as the function PRINT except that it binds *PRINT-PRETTY* to T, does not print a trailing space, and returns no values.

WRITE *object* **&key :stream :escape :base :pretty :level :length :lines :array :radix :circle :case :gensym**

This outputs *object* while allowing (but not requiring) the specification of output control values.

PP:INTERACTIVE-WRITE *object* **&optional** *stream*

This function prints *object* on *stream* under the control of the values set by PP:SET-INTERACTIVE-CONTROL-VARIABLES.

PP:SET-INTERACTIVE-CONTROL-VARIABLES &key :escape :base :pretty :level :length :lines :array :radix :circle :case :gensym

This function specifies the output control values to be used by PP:INTERACTIVE-WRITE.

PP:PP &optional *object stream*

This prints *object* (which defaults to the last object abbreviated due to PP:*PRINT-LINES* by PP:INTERACTIVE-WRITE) without abbreviation.

<FUNCTION> <RESUME>

Typing <FUNCTION> <RESUME> evaluates (PP:PP).

PP:FAST-FORMAT *destination control-string* **&rest** *args*

This macro is the same as the function FORMAT except that it is much more efficient and *destination* is required to either be a literal NIL or a literal T or a stream.

PP:DEFINE-LIST-PRINT-FUNCTION *symbol* (*list stream*) **&body** *forms*

This macro defines a list print-self function for lists whose CAR is *symbol.*

PP:UNDEFINE-LIST-PRINT-FUNCTION *symbol*

This function causes there to be no list print-self function associated with *symbol.*

PP:DEFINE-GENERALIZED-PRINT-FUNCTION *name* (*object stream*) *predicate* **&body** *forms*

This macro defines a generalized print-self function named *name.* The *predicate* is used to determine whether the generalized print-self function is applicable to a given object.

PP:GENERALIZED-PRINT-FUNCTION-ENABLED-P *name*

Determines whether the named generalized print-self function is enabled. SETF can be used to enable or disable a given generalized print-self function.

PP:WITH-GENERALIZED-PRINT-FUNCTION *name* **&body** *forms*

This macro evaluates the *forms* with the specified generalized print-self function enabled.

PP:DEFINE-FORMAT-DIRECTIVE *name* (*arg stream colon atsign* **&rest** *parameters*) **&body** *forms*

This macro defines a FORMAT directive which prints a FORMAT argument. The directive can be used by putting ~*package:name*\ in a FORMAT control string.

Summary of Variables

PRINT-PRETTY (default NIL)
When non-NIL this standard Common Lisp variable enables pretty printing.

PRINT-ESCAPE (default T)
When non-NIL this standard Common Lisp variable forces the printing of escape characters.

PRINLEVEL (default NIL)
When non-NIL, this standard Lisp variable limits the maximum nesting of logical blocks. (When Common Lisp compatibility is introduced this variable will be renamed *PRINT-LEVEL*.)

PRINLENGTH (default NIL)
When non-NIL, this standard Lisp variable limits the maximum number of sections printed in a logical block. (When Common Lisp compatibility is introduced this variable will be renamed *PRINT-LENGTH*.)

PP:*PRINT-LINES* (default NIL)
When non-NIL, this variable limits the number of lines printed by an outermost logical block.

PP:*PRINT-RIGHT-MARGIN* (default NIL)
When non-NIL, this specifies the line width to use for pretty printing.

PP:*PRINT-MISER-WIDTH* (default 40)
This specifies when logical blocks should be printed in miser mode.

Summary of Directives

The directive ~W (write object) uses the function WRITE to output the corresponding FORMAT argument without forcing the setting of any output control variables. The directive ~W takes the same parameters as ~A -- i.e., *~mincol, colinc, minpad, padchar*W.

~W Prints an argument following all output control variables.
~:W Forces pretty printing and suppresses abbreviation.
~@W Forces any padding to be inserted on the left.

There are three special directives for printing lists. Each of them prints parentheses around the output if and only if they are given an initial parameter of 1.

~\FILL\ Prints as many elements as possible on each line.
~\LINEAR\ Prints the elements all on one line or one to a line.
~,*c*\TABULAR\ Prints the elements in a table with column spacing *c*.

The directive ~!*prefix*~;*body*~;*suffix*~. (logical block) iterates over a list argument using *body* to print the elements of the list in a logical block. The *prefix* and *suffix* are printed before and after the body respectively.

~!...~. Creates a logical block and descends into a list argument.
~@!...~. Operates on all the remaining arguments.
~:!...~. Forces pretty printing and suppresses abbreviation.
~1!*body*~. Prefix and suffix default to "(" and ")" respectively.
~!*prefix*~@;~. Prefix printed on each line.
~!*body*~@. Body printed to fill the line width.

The indentation in a logical block is initially set to the column position of the first character after the prefix. The directive ~I (set indentation) is used to control the indentation within a logical block. If omitted, the parameter defaults to zero. When a logical block is printed in miser mode, all instances of ~I are ignored.

~*n*I Indentation set to the position of the first character after the prefix plus *n*.
~*n*:I Indentation set to the position of the directive plus *n*.

The directive ~_ (conditional newline) specifies a place where a newline can be inserted in a logical block. If a logical block is being printed in miser mode than a newline is inserted at every ~_ directive in it. Outside of a logical block, ~_ has no effect.

~_ Newline if the enclosing block is printed in multi-line mode.
~:_ Newline if the following block section will not fit on the end of the line.
~@_ Newline only if the enclosing block is printed in miser mode.

When pretty printing inside of a logical block, the directives ~% and ~& cause indentation (and the insertion of any per-line prefixes) in the same way as ~_.

Inside of a logical block, the standard FORMAT directive ~T is changed so that it tabs relative to the beginning of the containing logical block section instead of relative to the start of the line.

Inside of a logical block, the standard FORMAT directive ~^ is changed so that it checks for the argument list being reduced to a non-NIL atom.